Supporting continuous integration by mashing-up software quality information

Brandtner, Martin ; Giger, Emanuel ; Gall, Harald

Abstract: Continuous Integration (CI) has become an established best practice of modern software development. Its philosophy of regularly integrating the changes of individual developers with the mainline code base saves the entire development team from descending into Integration Hell, a term coined in the field of extreme programming. In practice CI is supported by automated tools to cope with this repeated integration of source code through automated builds, testing, and deployments. Currently available products, for example, Jenkins-CI, SonarQube or GitHub, allow for the implementation of a seamless CI-process. One of the main problems, however, is that relevant information about the quality and health of a software system is both scattered across those tools and across multiple views. We address this challenging problem by raising awareness of quality aspects and tailor this information to particular stakeholders, such as developers or testers. For that we present a quality awareness framework and platform called SQA-Mashup. It makes use of the service-based mashup paradigm and integrates information from the entire CI-toolchain in a single service. To evaluate its usefulness we conducted a user study. It showed that SQA-Mashup's single point of access allows to answer questions regarding the state of a system more quickly and accurately than standalone CI-tools.

DOI: https://doi.org/10.1109/CSMR-WCRE.2014.6747169

Supporting Continuous Integration by Mashing-Up Software Quality Information

Martin Brandtner, Emanuel Giger, and Harald Gall
University of Zurich, Department of Informatics, Switzerland
Email: {brandtner, giger, gall}@ifi.uzh.ch

***Abstract*—Continuous Integration (CI) has become an established best practice of modern software development. Its philosophy of regularly integrating the changes of individual developers with the mainline code base saves the entire development team from descending into *Integration Hell*, a term coined in the field of extreme programming. In practice CI is supported by automated tools to cope with this repeated integration of source code through automated builds, testing, and deployments. Currently available products, for example, Jenkins-CI, SonarQube or GitHub, allow for the implementation of a seamless CI-process. One of the main problems, however, is that relevant information about the quality and health of a software system is both scattered across those tools and across multiple views. We address this challenging problem by raising awareness of quality aspects and tailor this information to particular stakeholders, such as developers or testers. For that we present a quality awareness framework and platform called SQA-Mashup. It makes use of the service-based mashup paradigm and integrates information from the entire CI-toolchain in a single service. To evaluate its usefulness we conducted a user study. It showed that SQA-Mashup's single point of access allows to answer questions regarding the state of a system more quickly and accurately than standalone CI-tools.**

I. Introduction

A fundamental aspect of Continuous Integration (CI) according to Fowler is visibility: "*[...] you want to ensure that everyone can easily see the state of the system and the changes that have been made to it.*"[1]. The integration of a modern CI-toolchain within the development process of a software project is fully automated, and its execution is triggered after every commit. Developers or testers perceive the CI process most often only in case of a build break or test failure. In such a case, they get an automatically generated notification, for example, via email. A developer can then fix the problem using this information. This kind of exception-driven behavior helps to detect and fix problems as early as possibly during integration runs. Even in the absence of build or test failures, modern CI-tools generate a bulk of data with each commit. This data is scattered across the entire CI-toolchain and analyzing it, for instance, to monitor quality of a system, is a time consuming task. This can delay the rapid feedback cycles of CI and one of its major benefits is then not realized.

The need for an integration of the tools used in a CI-landscape is expressed in studies that address the information needs of software developers. Questions are, for example, *What has changed between two builds [and] who has changed it?* [2]. Typically the way to answer these kind of questions require two steps. First, a developer needs to know the dates of the respective builds. Second, these build dates can then be used to answer the question itself by investigating, e.g., commit logs, file-diff data, build details, issue details etc. However, to obtain the relevant information a developer must access several different tools and navigate through multiple views.

Additionally, the scope covered by modern CI-toolchains is not limited to build break and test failures. Quality measurements, such as code metrics, violations of coding guidelines, or potential bugs, are computed with each integration run. They provide an immediate feedback circuit for recently committed source code changes. Currently, this immediate feedback is limited to email notifications from each CI-tool due to the missing integration along a CI-toolchain. Integration approaches, such as the work of Singer et al. [3], address only tools used by developers during the creation of source code.

We derived seven specific tool requirements from the literature (see Table I) and implemented a proof-of-concept mashup for data integration and a front-end for its representation. Our framework and platform called *SQA-Mashup* is highly extensible and integrates information from various CI-tools such as BitBucket, GitHub, Jenkins-CI, and SonarQube. The graphical front-end of SQA-Mashup is role-based and supports tailored views associated to different stakeholders in the context of a software project. In this paper we use the term *stakeholder* to refer to developers and testers as these two roles are currently supported. However, we emphasize that our approach can be extended by including views for other stakeholders, such as software architects or project managers.

The developer view provides code-related measurements, such as code complexity etc.; the tester view provides testing-related measurements such as test coverage etc. To validate our approach we defined the following research question:

RQ: How do stakeholders perform in answering questions about software quality with SQA-Mashup compared to the use of standalone CI-tools?

We answered this research question with a user study that we conducted with 16 participants on the JUnit project. The results of our study show empirical evidence that an integrated platform has substantial advantages in terms of accuracy of answers and time needed to answer questions: information needs covered by an integration of CI-tools lead to more accurate answers in less time compared to the standalone use of CI-tools.

The remainder of this paper is structured as follows: We discuss related work in Section II. In Section III we briefly introduce the SQA-Mashup approach. The experimental design

/14/$31.00 © 2014 IEEE

of our user study is described in Section IV. In Section V we provide the results of the evaluation, and we discuss our findings in Section VI.

II. Related Work

The goal of our approach is to support the answering of common questions in the context of software evolution and continuous integration. We follow insights of prior research analyzing the information needs of developers to ensure that our set of questions is relevant and useful. We divide the field of the related work into the following areas: software evolution in general, information needs, and continuous integration.

Software evolution: Mens et al. [4] list a number of challenges in software evolution. One approach to reason about the evolution of software systems is *the integration of data from a wide variety of sources*. The Moose project [5] is a platform, which addresses the integration of data from different of data sources, such as source code repositories or issue trackers. FM3 is the meta-model used in Moose, which builds the base for an automatic evolution analysis. Another approach based on Semantic Web technologies was proposed by Li and Zhang [6]. Our approach tackles the integration differently since we do not use a meta-model, such as Moose, to analyze the evolution of a software system. Instead, we integrate data with what is already computed by state-of-the-art CI-tools.

Information needs: Aranda and Venolia [7] investigated source code histories in repositories to identify common bug fixing patterns. Breu et al. [8] extracted a catalog of questions from bug reports. They found that many questions are not explicitly stated in issues and therefore often not answered. Other studies conclude with an explicit catalog of questions asked by a group of stakeholders. LaToza and Myers [9] came up with a list of *Hard-to-answer questions about code*. Fritz and Murphy [2] provide a collection of developer's questions in combination with an information fragment model to answer these questions. Another group of studies examined the tasks of professionals during software comprehension. Roehm et al. [10] showed that developers follow different strategies to complete their work. Müller and Fritz [11] shifted the focus from software testers and developers to a more diverse audience such as requirements engineers and product line managers. They found that these stakeholders require multiple different artifacts to perform their daily activities. Their findings extend existing studies [2], [9] performed with software developers.

Continuous integration: Staff and Ernst reported on a controlled human experiment to evaluate the impact of continuous testing onto the source code quality [12]. The results of their experiment showed an improvement caused by the immediate feedback from the continuous testing framework. Their approach was implemented as plugins for Eclipse and Emacs. A similar experiment was conducted by Hurdugaci and Zaidman [13]. They implemented a plugin for Visual Studio, which *helps developers to identify the unit tests that need to be altered and executed after a code change* [13]. Both experiments showed that an immediate feedback from CI-tools fosters software quality during software development and maintenance. We base upon these results and aim at mashing up the proper information for the respective stakeholder.

III. Mashing-Up Software Quality Data

The goal of our approach is to integrate the information scattered across the CI-tooling landscape into a single Web-based service and to present it according to the information needs of different stakeholders. Our approach aims for a fast and easy way to analyze and communicate the state, health, and recent changes of a system.

In the following, we discuss our approach of data integration and presentation. We discuss tool requirements and list associated literature. We then present our mashup-based data integration approach that is inspired by Yahoo pipes.[1] Finally, SQA-Mashup, a proof-of-concept implementation of the approach is illustrated, followed by a description of its flexible Web-service API and role-based user interface.

A. Data Integration and Presentation Approach

The aim of our approach is *the integration of CI-data from various CI-tools* and *the presentation of this data according to the information needs of stakeholders*.

In modern CI-tools, the initial access to the data is facilitated by means of Web-Service interfaces. The challenge lies in the integration and interlinking of the data across the various data sources. The findings of Wasserman [14] about integration issues in computer-aided software engineering, such as the absence of standards for tool integration, are also applicable for CI-tools. We address this challenge with a *mashup-based approach*, which enables a flexible way of selecting specific data from multiple data sources and to integrate it into one consistent information space.

The mashup allows for a pipe-and-filter based processing of the integration steps. Every *integration pipe* in the SQA-Mashup service back-end is described as a series of pipe-and-filter steps (see Section III-C). The execution of such a pipe is triggered by a Web-service call.

Our approach builds on a service back-end, which integrates the data for a dynamic user interface composition based on information needs. We leverage this flexibility for a presentation which allows for a dynamic arrangement of aggregated CI-data. Every integration pipe of the SQA-Mashup service back-end can be visualized through a *widget* in the front-end. A widget in our context is a graphical representation of an integration pipe. A strict separation of concerns enables the visualization of one pipe output with different widgets, such as a list, a chart or a tree-map. We pushed this flexibility even further and enabled the creation of *views*. A view is an individual arrangement of widgets according to the needs of a stakeholder group or of an individual stakeholder. In the current implementation, we offer a view for developers and a view for testers. These two views differ in the kind and the granularity of the presented information. For example, the pipe with data about the unit tests is presented in the developer view as a chart of passed

[1] http://pipes.yahoo.com/pipes/

and failed test cases but in the tester view the same pipe is presented as a list with more detailed information.

B. Tool Requirements

We surveyed the literature for tool requirements and information needs of stakeholders during their work on source code. The work of Mockus et al. [15] and LaToza et al. [9] was chosen as starting point for our further literature survey. We investigated related work and looked for explicitly-stated and implicitly-stated tool requirements.

Table I lists requirements for the integration platform presented in this paper. The column *kind* indicates if a tool requirement is explicitly (E) stated in literature or derived (D). We categorized each tool requirement based on the context in which it was stated in the literature: change awareness (CA) and tool-design-related requirements, such as user interface (UI) and architectural design (AD).

TABLE I
TOOL REQUIREMENTS

ID	Tool requirement	Kind	Cat	Ref
R1	Be able to locate quality hot-spots in source code (e.g. low test coverage, high complexity)	D	CA	[9]
R2	Permit dynamic arrangement of information shown in the user interface.	D	UI	[2]
R3	Provide awareness of the activities of peers (co-workers).	D	CA	[7]
R4	Be able to discover immediately where changes occurred, when they were made, and who made them.	E	CA	[15]
R5	Provide an interactive visualization of a person's role (e.g., developer, tester, manager).	E	UI	[15]
R6	Be able to interoperate with other software engineering tools.	E	AD	[3]
R7	Permit the independent development of user interfaces (clients).	E	AD	[3]

The entries in Table I show that more recent literature has a strong focus on change awareness, whereas, traditionally the focus has been on architectural design and user interfaces. The reason for this can be found in recent technological innovations, from which service-based architectures emerged over the last decade. Nowadays, many CI-platforms offer Web-service based interface to exchange data with other systems. Tool requirement R6 reflects this functionality and requests for interoperability rather than a simple data exchange over pre-defined interfaces. Interoperability has to guarantee that exchanged data can be processed in each of the associated systems. Current CI-tools exhibit limited interoperability, because their service interfaces are a collection of flat data optimized for the associated user interface. A tight integration between service interface and user interface is contradictory to tool requirement R7. The Web-service interface of Jenkins-CI is an example for a Web-service optimized for the default front-end. Independent development of a user interface is possible if the data layer is designed in a generic way or offers configurable services. Our definition of configurable services is described in Section III-D.

A dynamic arrangement of UI-elements in state-of-the-art tools is possible but often restricted to one configuration per project. Personalized views according to a stakeholder role and personal preferences are not supported by CI-tools such as SonarQube. It is possible to address this issue with plugins, for example, to achieve a simple rearrangement of widgets for a personalized view. The capability to combine widgets from multiple views within one view is often restricted by the underlying architecture of a CI-tool. For example, a tool has one view on development-related data and a second view on testing-related data. The ability of a CI-tool to handle combined views is stated in tool requirement R2. A similar tool requirement, but yet on another level is stated through tool requirement R5. The dynamic level added through tool requirement R5 allows for interactive views based on a stakeholder's role. An interactive visualization adopts the view of a stakeholder based on latest changes in the data of a CI-toolchain and with respect to the affected stakeholder's role.

Agile development processes do iterations within the development and maintenance phase compared to other software development processes. More iterations within the same amount of time require the ability to respond faster to changes happening during an iteration. Particular CI-tools support agile software development but each single tool restricts the data presentation to its own data. Within each iteration a stakeholder has to navigate through every tool to become aware of changes since the last build. Tool requirement R4 addresses the awareness of changes within each build. A similar direction is stated by tool requirement R3. The difference between information need R3 and R4 is the point of view; R3 addresses the awareness between co-workers, whereas R4 addresses the awareness of changes within a certain period (for example, an iteration). R1 has a special focus on software quality measurements. Similar to R4, the aim of R1 is to become aware of changes.

C. Web-Service Integration

We use a mashup-based solution to collect and aggregate data from different CI-tools. Our mashup solution called *SQA-Mashup* consists of three data processing steps:

- *Fetch:* Expects a URI of a Web-service and returns the response document of the Web-service call.
- *Merge:* Expects two documents as input parameter and returns one document containing the data of both input documents.
- *Select:* Expects a document and one or more XPath expression(s) as input parameters. The outcome is one or more nodes of the document selected by one or more expression(s).

The advantage of such a mashup-based solution is the ability to combine each single step to a chain of processing steps. This allows for a dynamic and automatic processing of data from multiple data sources.

We use Representational State Transfer (RESTful) Web-services in combination with the JavaScript Object Notation (JSON) for the data collection, aggregation, processing, and propagation. JSON is a text-based data-exchange format with a low processing overhead and, in combination with RESTful Web-services, a de-facto standard in CI-tools.

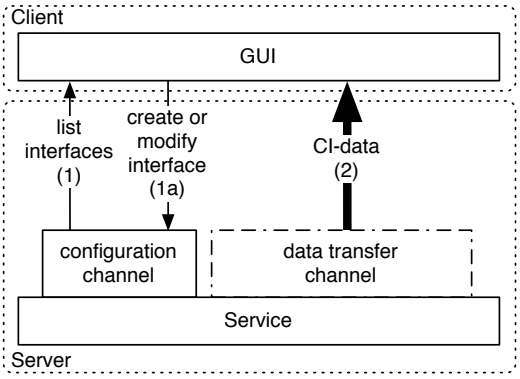

Fig. 1. Dynamic Web-service definition trough channels

The simplicity of JSON has two drawbacks, which affect the processing of our mashup-based implementation: there is no standardized query language and no context information, such as schemas or namspaces known from XML. Therefore, its not possible to automatically associate a JSON document to the context of a CI-tool, such as the build tool or testing tool context. We overcome the missing context by automatically deriving the context from the uniform resource identifier (URI) of a service call. We automatically add a key-value pair to each JSON document with the context information right after it is received from the Web-service of a CI-tool.

The missing schema and namespace definition within a JSON document also affects the possibilities to query information. For example, a key can occur multiple times in different positions within one document. We used JXPath[2] in combination with Jackson[3] and XPath to query for a key based on the surrounding structure in the JSON document. For example, we query for a key *metric-value* which has a sibling key called *metric-name* with the value *loc*. The result of this example query would be the value of the *lines of code* metric out of a document from the SonarQube Web-service.

D. Web-Service Presentation

The Web-service interface design plays an important role for satisfying tool requirement R7. The design of a Web-service interface aims for a clear separation of concerns and avoidance of redundant values in response data. This is different to a user interface that provides redundant data to allow a stakeholder to easily discover dependencies between data from different CI-tools based on the tool requirements R1-R4.

We overcome the divergence between the Web-service interface description and the visual representation with a client-server architecture and a twofold design of the Web-service interface (Figure 1). The twofold Web-service design consists out of a *configuration channel* and a *data transfer channel* for the communication between the service and a client.

The data transfer channel contains Web-services for the actual transfer of data from CI-tools to a client application, such as a graphical user interface (GUI). Each Web-service in the data transfer channel can be dynamically created, modified, or deleted during runtime. The logic to manage the data transfer channel is encapsulated in Web-services, which are part of the configuration channel. All Web-services used in the configuration channel are hard-coded and cannot be modified during runtime. The Web-services of the configuration channel represent an API which is the starting point for a client application. Every client application with access to this API is able to consume, adopt, and extend the data transfer channel. The twofold Web-service design enables an encapsulation of the complete integration logic and computation efforts into one place. This is especially interesting for concerns on change awareness such as tool requirements R2 and R4. An example is an application which visualizes the evolution of a software project based on data from CI-tools. Depending on a software project's history the amount of evolution data can be enormous. For such a large amount of data our approach enables a visualization client to query meta data, which can be used to limit the actual query size. Smaller response values which accord to the needed data format save computation power on the client device. The saved computation power can be used for enhancements such as richer or faster visualization.

E. Visual Representation

The visual representation presents the integrated data to a stakeholder according to tool requirement R5. Figure 2 depicts the *Developer View* and *Tester View* of the SQA-Mashup front-end. Both views have a similar appearance but present information for different stakeholders. For example, the second widget in the second row in Figure 2 visualizes the relative size of each source code package and the status of a quality measurement, such as rule compliance. The background color of each source code package depends on a pre-defined thresholds of a certain quality measurement. For example, a rule compliance above 90% leads to a green background color.

In case of the *Developer View* the background color indicates the rule compliance of each source code package according to, for example, the Java Code Conventions. The same widget is on the *Tester View* but the background color indicates the test coverage instead of rule compliance.

IV. Controlled User Study

We designed and conducted a controlled user study with 16 participants who had to solve nine software maintenance tasks. The participants were randomly assigned to either the control group or the experimental group. Each group consisted out of eight subjects and the control group confined the baseline of our study. This group had to use state-of-the-art CI-tools and a social coding platform: Jenkins-CI and SonarQube as the CI-tools and GitHub as the social coding platform. The experimental group had to use SQA-Mashup, which was implemented according to the tool requirements listed in Table I. More information about the study population is provided in Section IV-B and the nine tasks are listed in Table III.

We formulated three hypotheses (see Table II) and verified them through statistical tests based on the results of the user

[2]http://commons.apache.org/proper/commons-jxpath/
[3]http://jackson.codehaus.org/

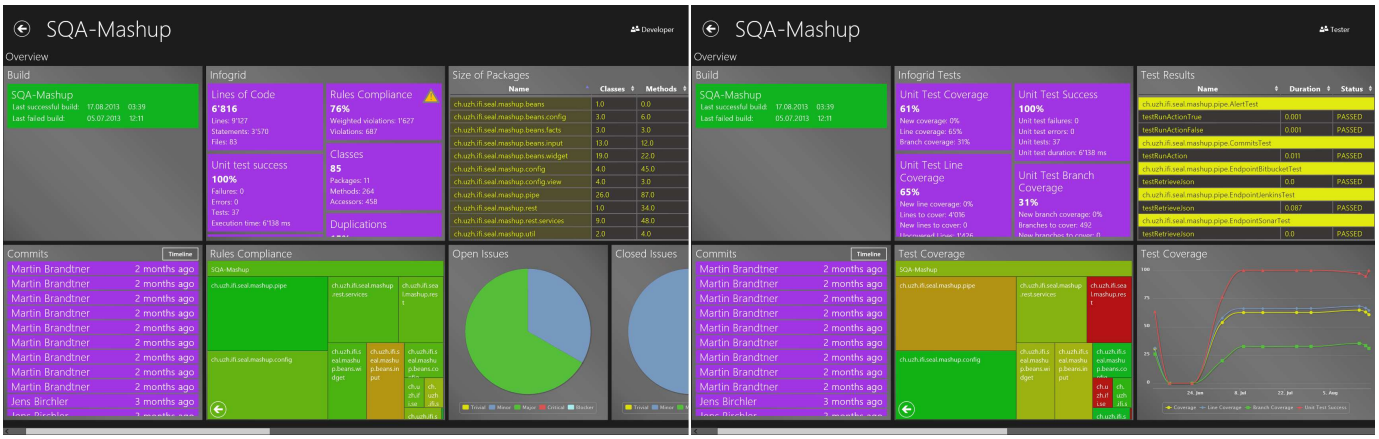

Fig. 2. SQA-Mashup - The screenshots illustrate the similar appearance of the Developer View (left) and Tester View (right).

study. The hypotheses address the total values of the score, the time, and the system usability score. The total score and time are summed up over all tasks per participant.

TABLE II
HYPOTHESES

ID	Null Hypothesis
$H1_0$	There is no difference in the total score per subjects between the experimental and the control group.
$H2_0$	There is no difference in the total time per subjects between the experimental and the control group.
$H3_0$	There is no difference in the total system usability score between the experimental and the control group.

A. Tasks - Nine Questions from the Literature

We exclusively selected tasks out of the software maintenance and evolution field which have been used in other studies or information need questions stated in literature. This is to ensure an objective comparison of our SQA-Mashup approach and the baseline approach with CI-tools.

We composed nine tasks which can be categorized along the domains *Program Comprehension*, *Testing*, and *Cross-Domain* questions. The tasks originate from the work of Aranda et al. [7], Fritz and Murphy [2], LaToza and Myers [9], Mockus et al. [16], and Wettel et al. [17].

Our aim was to preserve each task in its original structure and semantics. We applied a two-step procedure to achieve this aim. The first step was to replace all general expressions in a task with a specific entity of the software maintenance and evolution domain. For example, the question *"Is this tested?"* was refined to *"Is $Class_a$ tested?"*. In a second step we replaced placeholders such as $Class_a$ with entities from the software project used for the user study, in our case the *JUnit* project. We provide a short statement for each task to outline the reason why we selected it.

Table III lists the nine tasks with a short statement, categorized by their nature, and with references to the literature.

B. Study Setting

We ran the user study with 16 participants (S1-16). Each subject had at least five years of development experience in general and minimum three years development experience with Java. 14 participants were advanced graduate students. 11 out of these 14 graduate students worked part-time in industry. The remaining two participants were one post-doctoral researcher and one PhD student. The youngest participant is 22 years and the oldest 49 years old. The median age of the participants was 27.9 years. To avoid bias, the students were recruited from courses, which were held by people other than the main authors. The control group and the experimental group consist out of eight participants each. Every subject was randomly assigned to either one of both.

Our SQA-Mashup approach allows for an integration of data originate from a CI-toolchain. The tools used in a CI-chain can vary between software projects. Nevertheless, we had to select CI-tools for the baseline of our study. We decided to stay with two major open source developer communities and selected the CI-tools used in the Eclipse and Apache community. The Apache Software Foundation (ASF) provides Jenkins-CI and GitHub repositories for all projects under the ASF Development Process. SonarQube provides an instance[4] which regularly performs quality analysis of Apache projects. The Eclipse Foundation provides Hudson-CI and GitHub repositories for all projects under the Eclipse Development Process. Therefore, we decided to use Jenkins-CI, GitHub, and SonarQube as baseline CI-tools in the control group of our study. Hudson-CI and Jenkins-CI are similar in their usage as they are forks of the same codebase. The experimental group of our study had to use the SQA-Mashup front-end. The input data was gained from the same CI-tools instances used by the control group.

Each participant of our study had to solve nine tasks (Section IV-A) in the context of the JUnit project hosted on GitHub. JUnit [18] is a popular unit testing framework for source code written in Java. The project is under active development since a decade. The JUnit project team uses the built-in issue tracker of GitHub to manage bugs and feature requests. At the time of our study the JUnit project on GitHub had in total 1650 revisions, about 30k lines of code, 54 contributors, and 736 issues. We decided to select the JUnit project because of the mature stage

[4]http://nemo.sonarqube.org/

TABLE III
TASKS

ID	Task	Ref
Program Comprehension Domain		
T1	**Description.** How big is the source code of $Project_a$? **Statement.** Values such as lines of code, number of classes, and number of packages allow a first approximation of the project size and structure.	[9]
T2	**Description.** The three classes with the highest method complexity in $Project_a$? **Statement.** The refactoring of complex parts of source code leads to a better maintainability of software.	[17]
T3	**Description.** What is the evolution of the source code in $Project_a$? **Statement.** To understand a software, it is important to know about the evolution. One way to describe the evolution of a software is to look at testing coverage and coding violations over time.	[2]
Testing Domain		
T4	**Description.** Is $Class_a$ tested? **Statement.** Untested parts of source code does potentially increase the risk of bugs.	[9]
T5	**Description.** Which part of the code in $Project_a$ takes most of the execution time? **Statement.** The performance of a system can be influenced by a single piece of code. The execution time of unit test can be used to find such pieces with a weak performance.	[9]
T6	**Description.** Three packages with a test coverage lower than the overall test coverage in $Project_a$? **Statement.** *"Code coverage is a sensible and practical measure of test effectiveness."* [16]	[16]
Cross-Domain		
T7	**Description.** Between which builds changed the status of $Issue_a$? **Statement.** The status changes of an issue can be used to track recurring bugs with each build.	[7]
T8	**Description.** What have my coworkers been doing between $Date_a$ and $Date_b$? **Statement.** Quickly assessing the current status in a software project can support context switches between multiple projects.	[2]
T9	**Description.** What has changed between $Build_c$ and $Build_d$? Who has changed it? **Statement.** Finding changes which introduced a bug but still let the source code compile can cause substantial effort.	[2]

of the project. At the time of our study the commits indicated that most of the source code changes address bugs or refactoring tasks. Only a small set of commits introduced new features. We think it is therefore a good example for a software project which is in its maintenance phase.

The maximum time to work through all tasks of our study was 45 minutes with maximum of five minutes per task. After five minutes a participant had to stop to work and to go on with the next question. The execution of one study in total lasts approximate 40-60 minutes as there was no time restriction on the introduction and the feedback section.

C. Usability

We were interested of what the participants of our study think about the usability of the CI-tools they used to solve the tasks. We decided to use the popular *System Usability Scale* (SUS) by Brooke [19]. The SUS is a simple and easy understandable scheme which consists out of ten questions each rated on a scale from "Strongly Agree" to "Strongly Disagree." The subjects of the control group were asked to give a cumulative feedback for all CI-tools they used during the study and the experimental group rated SQA-Mashup.

As part of our user study each participant was asked to rate the *Difficulty* of each task right after solving it. We used a scale from "Very difficult" to "Very easy" in which a participant provides a personal opinion. This additional information should allow us to sharpen our understanding of the usability rating.

D. Performing the Study

We performed the user study in four sessions in the lab of our research group in Zurich. The setup for the experimental group was a computer with Windows 8 and the SQA-Mashup front-end installed on it. The setup for the control group was a standard Web-browser such as Mozilla Firefox or Google Chrome. The subjects of the experimental group started at the project page of JUnit in the SQA-Mashup front-end. Participants of the control group started with three Web-browser windows. One window for Jenkins-CI, one for SonarQube, and one for the project page on GitHub.

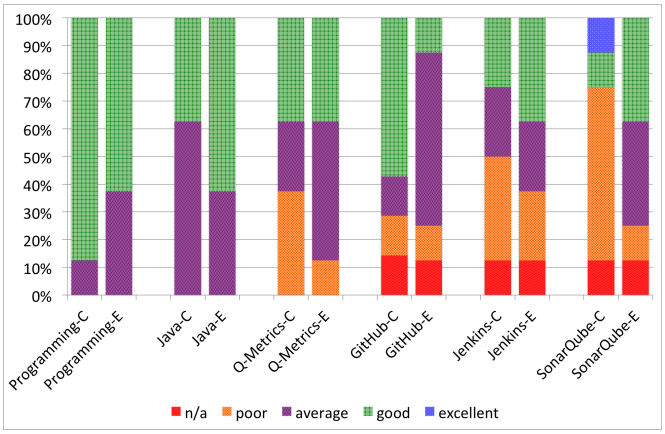

Fig. 3. Self-assessment - (C)ontrol group and (E)xperimental group

At the beginning of every session we explained the structure of the study in addition to the written explanation on the questionnaire. The participants solved the tasks independently and the time restriction was enforced with individual stop-watches. We asked the participants of the user study to do a self-assessment of their Programming skills as-well-as their working experience with the CI-tools used in the study. Figure 3 depicts the self-assessment results of the control group and the experimental group. Only one participant had no working experience with any of the used platforms such as GitHub,

Jenkins, and SonarQube. All others had experience with at least two out of these three platforms. The self-assessment results of the experimental group and control group were statistically not significantly different.

E. Data Collection

The answers of the study participants were manually pre-processed for an analysis of the data with R [20].

The questionnaire of the user study consists of single choice questions, open questions, and rating scales for usability related questions. Every answer of the questionnaire was mapped to a numeric representation to run a statistical analysis on it. We used the following rules for the mapping:

- Rating scales: The index of the selected entry is the numerical representation, starting with zero for the first entry.
- Single choice and open questions: A grading scheme based on points with 3 points for a correct answer, 2 points for a partly correct answer (for open questions only), 1 point for a wrong answer, and 0 points for no answer.

We graded an answer of an open question as partly correct if it was correct but incomplete. For example, if the correct answer consists of two artifacts but only one was found by a participant. In case of an open question with only one correct answer the grading is the same as for single choice questions.

V. Empirical Results

In this section we statistically examine the results of the user study with respect to the hypotheses as listed in Table II. In Section V-A we analyze $H1_0$ and $H2_0$ using the aggregate results: total score per subject, total time per subject, and the ratio of total score to total time per subject aggregated over all nine tasks. In Section V-B we analogously analyze $H1_0$ and $H2_0$ on the level of each individual task. The results of the SUS and the difficulty ratings are discussed in Section V-C.

A. Analysis & Overview of Aggregated Results

Total Score: Table IV lists the total score per subject aggregated over all tasks for both, the experimental and control group. Each of the nine tasks was graded with 3 points if solved correctly (see Section IV-E), so the maximum score a participant could achieve was 27. The median of the total scores was 20.0 points in the control group and 25.5 points in the experimental group. In other words, this median difference of 5.5 points means that the experimental group outperformed the control group on average by 20.4% (=5.5/27) when referring to the total score per subject. The minimum total score per subject was 18 points (S12) and 17 points (S4), the maximum was 24 points (S10) and 26 points (S1-3,14) in the control and the experimental group, respectively. None of the participants solved all tasks correctly. One outlier (S12) with 12 points in the control group solved only four tasks - but those correctly. We could not observe any particularly obvious reasons, such as technical difficulties, tool failure, or misunderstandings of the tasks, for this comparably low score and hence did not exclude subject S12 from our analysis.

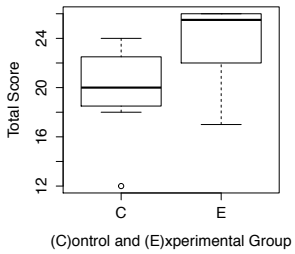

Fig. 4. Total Score

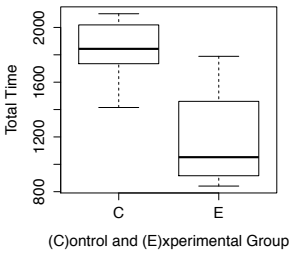

Fig. 5. Total Time

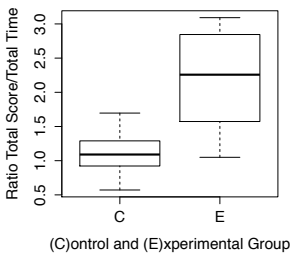

Fig. 6. Ratio Total Score/Time

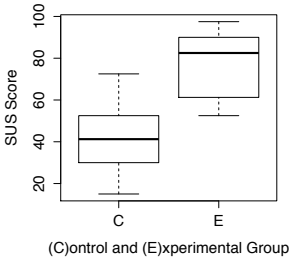

Fig. 7. SUS Score

The box-plot of the total score per subject (see Figure 4) of the experimental group shows that the 0.75 quantile and the median are close and almost the same as the upper whisker of the plot. This indicates a skewed distribution. Furthermore, Q-Q plots and Shapiro-Wilk tests showed significant evidence against normality in the data. We made similar observations regarding the other quantities (see Figures 4-7) although the departures are less. Since our group sample sizes are below the commonly accepted rule of thumb (at minimum 30 to 40) we chose a non-parametric, independent-two-samples procedure, i.e., Mann-Whitney-Wilcoxon (MWW), to test all hypotheses. Loosely speaking, MWW determines if the values in the experimental group are (significantly) different by comparing the mean ranks of the values in both groups. If both groups come from identical distribution they have equal chances for high and low ranks, and there is no difference in their mean ranks. Note, that the actual test itself is based on the U statistic [21]. The box-plots further reveal that distributions of the values between the experimental and the control group might exhibit dissimilar shapes. In this case, MWW is interpreted as a test of stochastic equality [21] rather than a test of medians.

Under the above considerations $H1_0$ (see Table II) states that the mean ranks of total score per subject of the two groups are equal. A two-sided MWW test resulted in a p-value of 0.038 and 0.95 (non-parametric) confidence interval bounds of [1, 7]. This gives us significant evidence against $H1_0$: The total scores per subject in the experimental group (mean rank = 11) are significantly different than for the control group (mean rank = 6.0). In other words, based on the p-value and direction of the difference (as indicated by the interval bounds) we find significant evidence that the subjects of the experimental group score higher.

Total Time: Table IV lists the total time per subject aggregated over all tasks for both, the experimental and control

group. The median of the total time was 30.7 minutes in the control group and 17.5 minutes in the experimental group. This median difference of 13.2 minutes means that the experimental group outperformed the control group on average by 29.3% (=13.2/45) when referring to the total time per subject. The maximum total time per subject was 35 minutes (S11, S12), the minimum was 14 minutes (S14) and 14.2 minutes (S1) in the control and the experimental group, respectively. However, none of the participants reached the time limit of 45 minutes.

$H2_0$ (see Table II) states that the mean ranks of total time per subject of the two groups are equal. A two-sided MWW test resulted in a p-value of 0.003 and (non-parametric) 0.95 confidence interval bounds of [-977, -308]. This gives us significant evidence against $H2_0$: The total time per subjects in the experimental group (mean rank = 5.12) is significantly different than for the control group (mean rank = 11.88). Based on these results we find strong evidence that the subjects of the experimental group could solve the tasks in less time.

Ratio of Total Score to Total Time: Table IV lists the total score to total time ratio (st-ratio) per subject aggregated over all tasks for both, the experimental and control group. This value was multiplied by 100 to aid readability. The median of the st-ratio was 1.09 in the control group and 2.26 in the experimental group. In other words, this median difference of 1.17 means that the experimental group outperformed the control group by about plus 0.7 points (1.17/100*60) per minute.
A two-sided MWW test resulted in a p-value of 0.005, (non-parametric) 0.95 confidence interval bounds of [0.363, 1.798], and mean ranks of 11.75 and 5.25, for the experimental and control group.

TABLE IV
RESULTS PER SUBJECT - MEDIAN AND STANDARD DEVIATION (SD)

Subject	Group	Total Score	Total Time	Score/Time * 100
S1	E	26	853.0	3.05
S2	E	26	984.0	2.64
S3	E	26	1789.0	1.45
S4	E	17	1620.0	1.05
S5	C	22	1760.0	1.25
S6	C	23	1730.0	1.33
S7	C	19	1927.0	0.99
S8	E	25	980.0	2.55
S9	E	22	1300.0	1.69
S10	C	24	1415.0	1.70
S11	C	18	2097.0	0.86
S12	C	12	2100.0	0.57
S13	E	22	1120.0	1.96
S14	E	26	841.0	3.09
S15	C	20	1740.0	1.15
S16	C	20	1940.0	1.03
median	C	20.0	1843.5	1.09
median	E	25.5	1052.0	2.26
SD	C	3.73	226.88	0.36
SD	E	3.24	355.06	0.76

B. Task Analysis

The aggregated results in the previous Section V-A show a significance evidence that the subjects using our SQA-Mashup approach tend to solve the tasks more correctly in less time compared to subjects using the given set of baseline tools. Correctness in our context is defined as the score achieved by a participant: A higher score means a higher correctness. The goal of this section is to further investigate where the advancements come from, and we break down the analysis of the global hypotheses H1 and H2 into individual tasks.

Due to the same reasons as on the aggregate level (non-normality and sample size below 30) the analyses on each task were performed with the MWW procedure. However, because of their post-hoc character we applied the Bonferroni-Holm [22] correction to the resulting p-values of the MWW tests on task level. This correction counteracts the problem of multiple hypotheses testing by controlling the family-wise error rate.

TABLE V
RESULTS PER TASK

	Score (median)			Time (median)		
Task	Control	Experim.	Δ	Control	Experim.	Δ
T1	3.0	3.0	0.0	67.5	40.0	27.5
T2	3.0	3.0	0.0	112.0	67.5	44.5
T3	1.0	2.0	1.0	285.0	225.0	60.0
T4	3.0	3.0	0.0	130.0	53.5	76.5
T5	1.0	3.0	2.0	240.0	50.0	190.0
T6	3.0	3.0	0.0	125.0	79.0	46.0
T7	0.5	2.0	1.5	300.0	300.0	0.0
T8	3.0	3.0	0.0	300.0	175.0	125.0
T9	3.0	3.0	0.0	300.0	150.0	150.0

1) Task T1 - Learning about project size: All participants in both groups were able to find this information and solved the task correctly. Moreover, a large adjusted p-value, such as $p>0.2$, clearly indicates that the observed time difference between the two groups is not significant.

Information about the project size is readily available in a particular view of the CI-toolchain. Hence not surprisingly, all participants of control group were able to write down the correct answer. We hypothesize that the single point of access for all information as provided by SQA-Mashup allows to spot the project size immediately and might offered the experimental group a head start. However, currently the results are not statistically conclusive and further work is needed on that issue, for instance, including additional participants.

2) Task T2 - Finding exceptionally complex source code: Every participant of both groups solved the task correctly. With respect to time a large adjusted p-value, such as $p>0.2$, shows that the measured time difference between both group is not significant. Again, analyzing additional subjects increases power of the test and would give more confident estimates and insights whether the observed times differences of such magnitudes in this task are due to chance.

3) Task T3 - Leaning about quality evolution: This task was solved with a significantly higher correctness (adjusted p-value of 0.025) by the experimental group but not necessarily faster (large adjusted p-value of $p>0.2$).

Task T3 captures an essential activity of CI, rigorously monitoring the state and quality of a software system over time [23]. However, to facilitate a comprehensive view of the state of the system, different quality indicators, such as test coverage, code metrics, test failures, or coding violations, must be considered. The subjects of the control group had

to manually search through the individual CI-tools to locate this information. Furthermore, they faced the difficulty that each tool uses different time periods and scales to calculate and visualize information. We generally noticed that only two subjects in the control group provided a partially correct answer, the remaining six subjects answered the task wrong. We interpret the results that a consistently integrated interface can foster project visibility and helps to encourage team awareness with respect to different software quality aspects.

4) Task T4 - Learning about test coverage: 14 participants solved the task correctly. Only one participant out of each group gave a wrong answer. With respect to time a large adjusted p-value, such as p>0.2, shows that there is no significant difference between the two groups.

5) Task T5 - Finding high execution times: This task was solved with a significantly higher correctness (adjusted p-value of 0.040) by the experimental group. We found a certain evidence (adjusted p-value of 0.073) that less time was needed.

While the experimental group likely benefited from the integrated view the subjects of the control group might have struggled with the fact that multiple CI-tools provide redundantly the same information about execution times.

6) Task T6 - Finding test coverage below average: Only one participant was not able to solve the task correctly. Moreover, with respect to time a large adjusted p-value, such as p>0.2, shows that the measured difference is not significant.

7) Task T7 - Relating build information to issues: This task was solved correctly by only two subjects of the control group and four subjects of the experimental group. Generally, this task seemed time-consuming and difficult since 75% of all participants reached the time limit of five minutes and over 50% rated it as "Very Difficult" (see Figure 8).

8) Task T8 - Learn about activities of coworkers: This task was solved significantly faster (adjusted p-value of 0.050) by the experimental group but not necessarily with higher correctness (large adjusted p-value).

The participants of the control group mentioned in their written feedback that they had difficulties in finding a specific view corresponding to the time period which we asked for in the user study. SQA-Mashup, on the other hand, offers a coherent, chronological view of developers and their activities.

9) Task T9 - Learn about activities between builds: This task was solved significantly faster (adjusted p-value of 0.033) by the experimental group but not necessarily with higher correctness (large adjusted p-value).

Task 9 is similar to task T8 from the perspective of builds. We expected a decrease in time to solve the task for both groups compared to task T8 because of learning effects and since builds are the central aspect of CI-tools. Only the median time of the experimental group decreased to 150 seconds, the median time of the control group remained high at 300 seconds. The scores increased in both groups.

C. Participant Ratings

We asked each participant to complete the SUS questionnaire and to rate the *Difficulty* of each task (Figure 8).

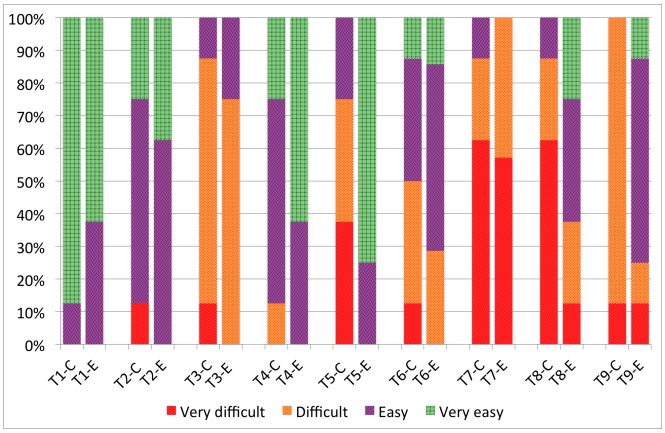

Fig. 8. Difficulty Rating - (C)ontrol group and (E)xperimental group

The result of a SUS is a score between 0 and 100 to represent the usability of a system. In our paper the score was calculated based on the original scoring scheme of Brooke [19] as follows: The contribution of every item with an odd index equals the score minus one, the contribution of every item with an even index is calculated as five minus the score. Finally, the sum of the resulting contributions is multiplied by 2.5. Figure 7 depicts a box-plot based on the SUS scores of the control and the experimental group. The usability median of the control group is 41.25 compared to 82.5 of the experimental group. Note, it is by chance that the usability measured in the experimental group is exactly twice of the control group's usability. The minimum whisker of the experimental group with 52.5 is slightly above the 0.75 quantile of the control group with a usability of 48.75.

$H3_0$ (see Table II) states that the mean ranks of the usability scores per subject of the both groups are equal. A two-sided MWW test resulted in a p-value of 0.004 and 0.95 confidence interval bounds of [15, 55]. This gives us significant evidence against $H3_0$: The usability score per subject in the experimental group (mean rank = 11.94) is significantly different from the score per subject in the control group (mean rank = 5.06). Based these findings there is strong evidence that the subjects of the experimental group perceived the integrated SQA-Mashup front-end more "user friendly".

Next, we describe differences in the individual task ratings regarding *Difficulty* between the control group and experimental group (see Figure 8). Two-sided MWW tests on each individual task rating resulted in large p-values. Noticeable differences are the ratings of the control group on task T2, T3, T5, and T6. In all cases the control group rated the three tasks as "Very difficult" in difference to the experimental group.

D. Discussion of the Results

Overall we found evidence that the participants of the experimental group solved the tasks of our study in less time and with a higher correctness. The SQA-Mashup approach was rated as more "user friendly" than the given baseline tools reflected by a significantly higher SUS score.

When analyzing these differences on the level of the individual tasks we found major insights in the benefits of

monitoring the CI-process and the capabilities of existing CI-tools. On the one hand, regarding single aspects of software quality, such project size (T1) or code complexity (T2), existing CI-tools provide already good support. The participants of both groups achieved a high correctness, and we could not observe a significant time gain in either of the two groups.

On the other hand, we found evidence that monitoring software quality during CI can substantially benefit from an integrated view. This is particularly the case when the required information is scattered over multiple CI-tools and displayed in different scales and units, for instance, as it is the case for tasks T3 or T8. The subjects of the control group had to search the CI-toolchain and then manually combine the information obtained from the individual tools. This seems to be a disadvantage compared to the experimental group that was presented a single interface that integrates all relevant quality information. Therefore, we see potential for such integrated approaches when multiple facts need to be merged to answer an aggregated quality aspect of a software system.

E. Threats to Validity

Unequal distribution of experience in terms of development experience and experience with the used tools can bias the results of a user study. We addressed this issue by a random assignment of the participants to a group. The analysis of both experiences with the MWW test indicated no significant difference of the two populations.

The *number of participants* might have influenced the drawn conclusion. Many results had a clear tendency but they were not statistically significant. A greater number of participants might help to draw a clear conclusion for those results.

Generalizability of the results. This threat is in respect to the participants of our study which were students and not professional developers. We tried to overcome this by selecting students with development experience in industry.

The *authenticity of the tasks* for the daily work of a software developer. The task selection of our user study is only a small extract of tasks which are done during software maintenance. We selected tasks from literature to avoid any favor in the direction of the baseline tools or SQA-mashup.

VI. Conclusions

The chance to descend into *Continuous Integration Hell* increases with each CI-tool added to a CI-toolchain. We developed SQA-Mashup, which dynamically integrates data from various CI-tools and tailors the information for stakeholders.

We deducted information needs and tasks from related works for the design and the evaluation of our proof-of-concept implementation. The integrated SQA-Mashup approach can (1) foster project visibility, (2) encourage team awareness, and (3) help to avoid the spreading of inconsistent information. The results of a user study showed that participants solved tasks with 21.6% higher correctness and 57% faster with SQA-Mashup compared to standalone CI-tools.

Next, we will tackle the scattering of information over multiple branches and the challenge of different scales and units in the visualizations. We currently work on a timeline-based user interface to facilitate software evolution monitoring.

Acknowledgment

We gratefully acknowledge the financial support from the Swiss National Science Foundation for our project "SoSYA - Systems of Systems Analysis" (Project No. 200021_132175). We are also grateful to our students Jens Birchler, Stefan Hiltebrand, and Hannes Tresch who implemented the first prototype of our SQA-Mashup tool.

References

[1] M. Fowler, "Continuous integration," 2006. [Online]. Available: http://www.martinfowler.com/articles/continuousIntegration.html

[2] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," ser. ICSE, Cape Town, 2010, pp. 175–184.

[3] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," ser. CASCON, Toronto, 1997, pp. 21–36.

[4] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," ser. IWPSE, Lisbon, 2005, pp. 13–22.

[5] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of moose: an agile reengineering environment," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 1–10, 2005.

[6] Y.-F. Li and H. Zhang, "Integrating software engineering data using semantic web technologies," ser. MSR, Waikiki, 2011, pp. 211–214.

[7] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," ser. ICSE, Washington, 2009, pp. 298–308.

[8] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users," ser. CSCW, Savannah, 2010, pp. 301–310.

[9] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU, Reno, 2010, pp. 8:1–8:6.

[10] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" ser. ICSE, Piscataway, 2012, pp. 255–265.

[11] S. C. Müller and T. Fritz, "Stakeholders' information needs for artifacts and their dependencies in a real world context," ser. ICSM, Eindhoven, 2013.

[12] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 76–85, 2004.

[13] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," ser. CSMR, Szeged, 2012, pp. 11–20.

[14] A. Wasserman, "Tool integration in software engineering environments," in *Software Engineering Environments*, ser. Lecture Notes in Computer Science, 1990, vol. 467, pp. 137–149.

[15] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," ser. ICSE, Orlando, 2002, pp. 503–512.

[16] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," ser. ESEM, Washington, 2009, pp. 291–301.

[17] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," ser. ICSE, Waikiki, 2011, pp. 551–560.

[18] P. Louridas, "Junit: unit testing and coiling in tandem," *Software, IEEE*, vol. 22, no. 4, pp. 12–15, 2005.

[19] J. Brooke, "Sus-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, p. 194, 1996.

[20] R Development Core Team, *R: A Language and Environment for Statistical Computing*, Vienna, 2011.

[21] H. Mann and D. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[22] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[23] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.